Tutoriat SO 10: Virtual Memory

January 9, 2026

Contents

1 Background

Virtual memory is a layer of abstraction that gives the illusion of a lot of contiguous memory available to each process, when in reality the available physical memory might be low. This idea was discussed when pagination was introduced though pagination is just an implementation of virtual memory.

So far we assumed the whole processes needs to be in memory in order to execute it's instructions, but we could also take only parts of the processes and request more as needed. Here are a few arguments as to why:

Motivation for Partial Program Loading

Programs often contain code to handle unusual error conditions. Since these errors seldom, if ever, occur in practice, this code is almost never executed.

Arrays, lists, and tables are frequently allocated more memory than they actually require. For example, an array may be declared as 100×100 elements even though it is rarely larger than 10×10.

Certain options and features of a program may be used only rarely. For instance, the routines on U.S. government computers that balance the budget have not been used in many years.

Even in cases where the entire program is needed, it may not all be required at the same time.

The ability to execute a program that is only partially loaded into memory would provide several important benefits:

- A program would no longer be constrained by the amount of available physical memory. Users could write programs for an extremely large *virtual address space*, simplifying the programming task.
- Because each program would require less physical memory, more programs could run concurrently. This would increase CPU utilization and system throughput, without increasing response time or turnaround time.
- Less I/O would be required to load or swap portions of programs into memory, allowing each program to execute faster.

2 Demand Paging

This idea of fetching more of the process only if needed is called demand paging. Imagine we have a menu with 4 options, loading the code for all of the options would be a waste if we can only choose one.

This is implemented using the valid/invalid bit scheme as depicted in the following image:

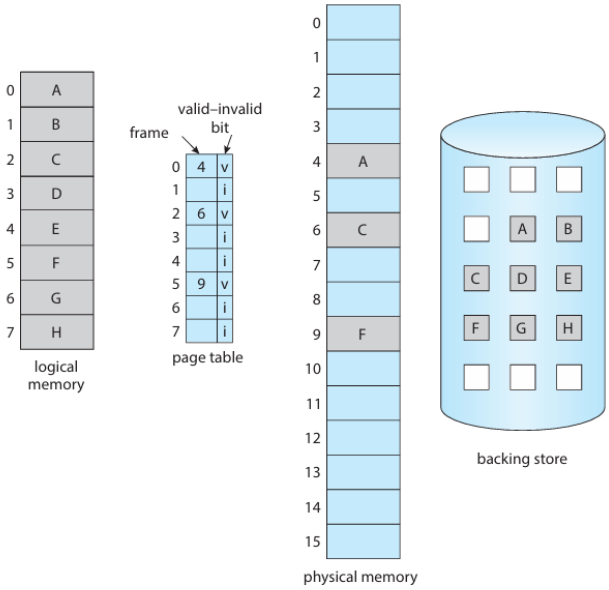

Figure 1: Demand Paging Implementation

The page is valid if it is mapped to frame in memory and invalid if either the page is outside the process's address space(bad) or the page is indeed in it's address space but not in main memory(only on the disk).

Accessing an invalid page creates a page fault. The paging hardware will then send interrupt to the operating system which will then bring the desired page from the disk to main memory.

Here is the whole algorithm:

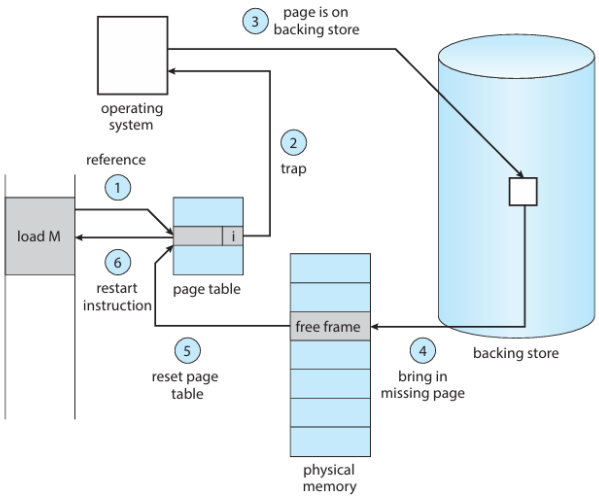

Figure 2: Steps in handling page fault

When a page fault occurs, the operating system takes the following steps:

1. The operating system checks an internal table (usually stored in the process control block) for the process to determine whether the memory reference was valid or invalid.

2. If the reference was invalid, the operating system terminates the process. If the reference was valid but the page has not yet been brought into memory, the operating system proceeds to load the page.

3. A free frame is located, for example by selecting one from the free-frame list.

4. A secondary storage (disk) operation is scheduled to read the desired page into the newly allocated frame.

5. When the disk read operation completes, the operating system updates the internal table associated with the process and the page table to indicate that the page is now resident in physical memory.

6. The instruction that was interrupted by the page fault trap is restarted. The process can now access the page as if it had always been in memory.

Neccessary Hardware: The hardware neccessary to implement demand paging are the page table and the disk(especially the swap space discussed more in Tutoriat 9 which is a special location on the disk that stores processes of parts of processes that need to be "swaped" back into main memory, maybe due to high memory demand)

One instruction, multiple page faults Demand paging can add more overhead to the system as for example a instruction could cause multiple page faults at once(the instruction an operands could live on different pages). This does not usually happen because processes have locality of reference(accesses to memory locations that are close to each other).

Restarting Instructions After a Page Fault We also have to restart the processes exactly from where it left of before the page fault. Before(and after) each page fault a context switch occurs so in most cases restarting is easy.

If a page fault occurs during an *instruction fetch*, the system can simply restart execution by fetching the instruction again. However, if a page fault occurs while fetching an *operand*, the instruction must be fetched and decoded again before the operand can be accessed.

Worst-Case Example As a worst-case example, consider a three-address instruction such as `ADD`, which adds the contents of memory location A to B and stores the result in C. The execution of this instruction consists of the following steps:

1. Fetch and decode the instruction (`ADD`).
2. Fetch operand A.
3. Fetch operand B.
4. Add A and B.
5. Store the result in C.

If a page fault occurs during step 5, while attempting to store the result in C (because C resides on a page that is not currently in memory), the operating system must retrieve the required page from secondary storage, update the page table, and then restart the instruction.

Restarting the instruction requires fetching and decoding the instruction again, fetching both operands again, and performing the addition again. Although some work is repeated, the amount of repeated work is limited to less than one complete instruction and occurs only when a page fault happens.

3 Free Frame List

Just like we used free block lists for the disk, we do the same for free frames. When the system first starts this list is full and as time goes on it shrinks until it hits a certain threshold after which it needs to be repopulated(probably using swapping).

Note: Frames are zeroed (their content is set to 0) before being allocated to a process to avoid data leaks.

4 Copy on Write (CoW)

Fork creates a new process that is the copy of the parent. Traditionally, fork also duplicated the pages of the child process. Taking into account fork is usually followed by exec this can be a waste of time.

For this CoW was introduced which only copies pages if either process tries to write to them. Initially both processes share the pages of the parent(which are now marked as copy on write) and with each write to a new page a copy of it is created. Pages that are read-only(like the code section) can be shared between the two processes(much like threads do).

Assume we have two processes(one is the child of another). Initially they share everything.

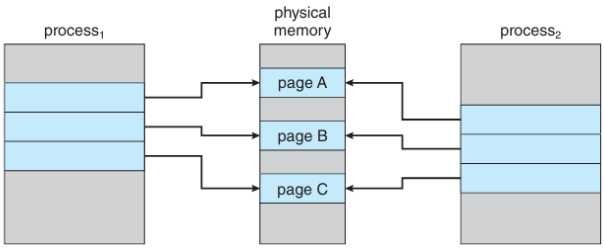

Figure 3: Cow example before

This is what happens when process 1 modifies page C.

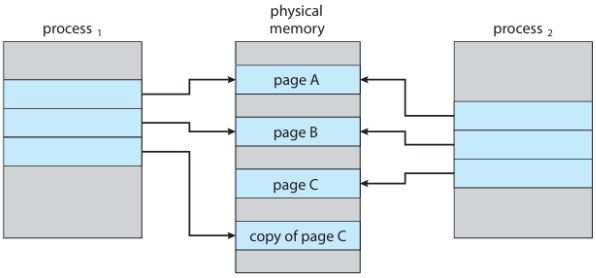

Figure 4: Cow example after

There is also a variation of fork called vfork(virtual fork) that does not set the pages as copy on write and just shares everything between the two processes, with the addition that it first suspends the parent. This is usually used when you want the fork+exec combination to perhaps get the extra benefit of not marking the pages as copy on write.

5 Page replacement

Page replacement takes the following approach. If no frame is free, we find one that is not currently being used and free it. We can free a frame by writing its contents to swap space

and changing the page table (and all other tables) to indicate that the page is no longer in memory. We can now use the freed frame to hold the page for which the process faulted.

We modify the page-fault service routine to include page replacement:

1. Find the location of the desired page on secondary storage.
2. Find a free frame:
   (a) If there is a free frame, use it.
   (b) If there is no free frame, use a page-replacement algorithm to select a victim frame.
   (c) Write the victim frame to secondary storage (if necessary); change the page and frame tables accordingly.
3. Read the desired page into the newly freed frame; change the page and frame tables.
4. Continue the process from where the page fault occurred.

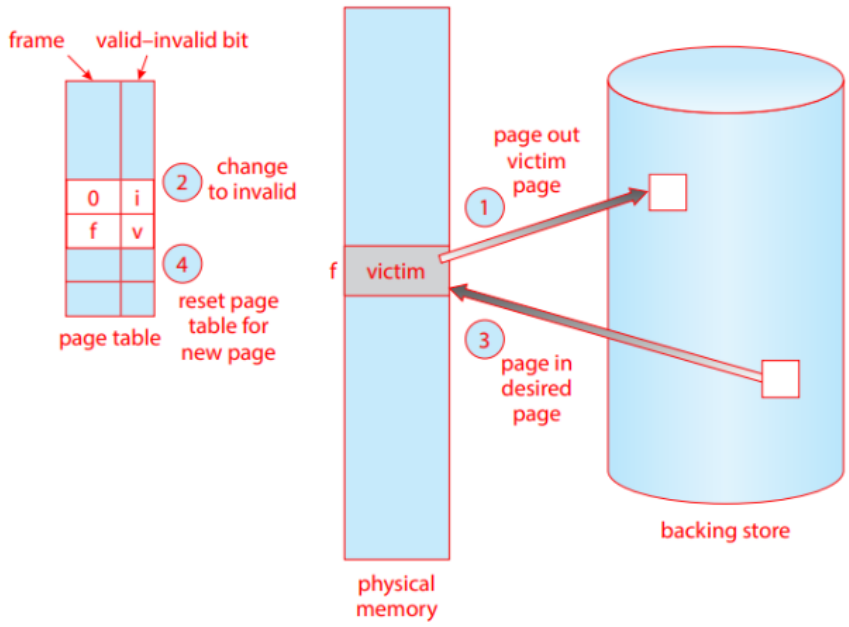

Figure 5: Page Replacement Steps

It's important to find this "victim" efficiently, so below we will discuss some algorithms to help us find the best solution.

5.1 FIFO

A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen.

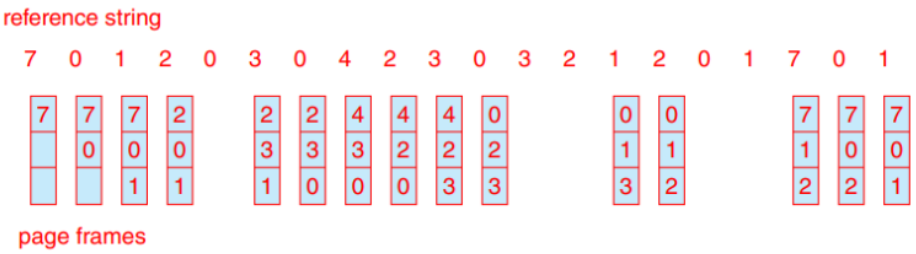

Figure 6: FIFO Page Replacement Example

To illustrate the problems that are possible with a FIFO page-replacement algorithm, consider the following reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5. Notice that the number of faults for four frames (ten) is greater than the number of faults for three frames (nine). This most unexpected result is known as Belady's anomaly: for some page-replacement algorithms, the page-fault rate may increase as the number of allocated frames increases.

We would expect that giving more memory to a process would improve its performance. Belady's anomaly was discovered as a result.

5.2 Optimal

One result of the discovery of Belady's anomaly was the search for an optimal page-replacement algorithm— the algorithm that has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly. Such an algorithm does exist and has been called OPT or MIN. It is simply this: Replace the page that will not be used for the longest period of time.

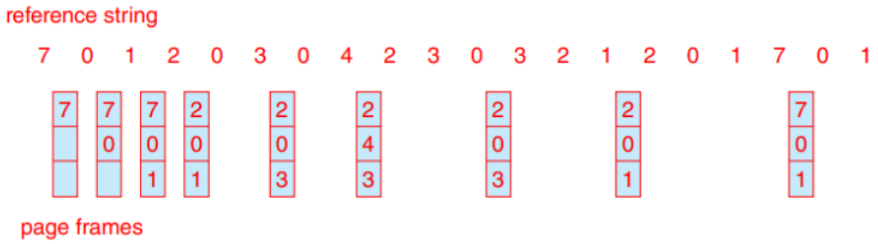

Figure 7: Optimal Page Replacement Example

Unfortunately, the optimal page-replacement algorithm is difficult to implement, because it requires future knowledge of the reference string.

5.3 LRU

If we use the recent past as an approximation of the near future, then we can replace the page that has not been used for the longest period of time. This approach is the least recently used (LRU) algorithm.

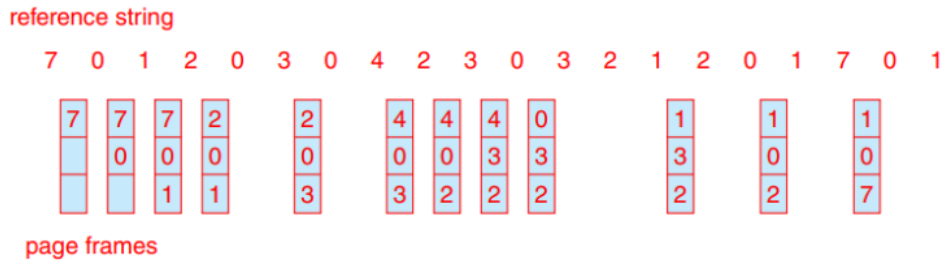

Figure 8: LRU Page Replacement Example

6 Thrashing

Consider what occurs if a process does not have "enough" frames— that is, it does not have the minimum number of frames it needs to support pages in the working set. The process will quickly page-fault. At this point, it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away.

Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately. This high paging activity is called thrashing. A process is thrashing if it is spending more time paging than executing. Thrashing results in severe performance problems.

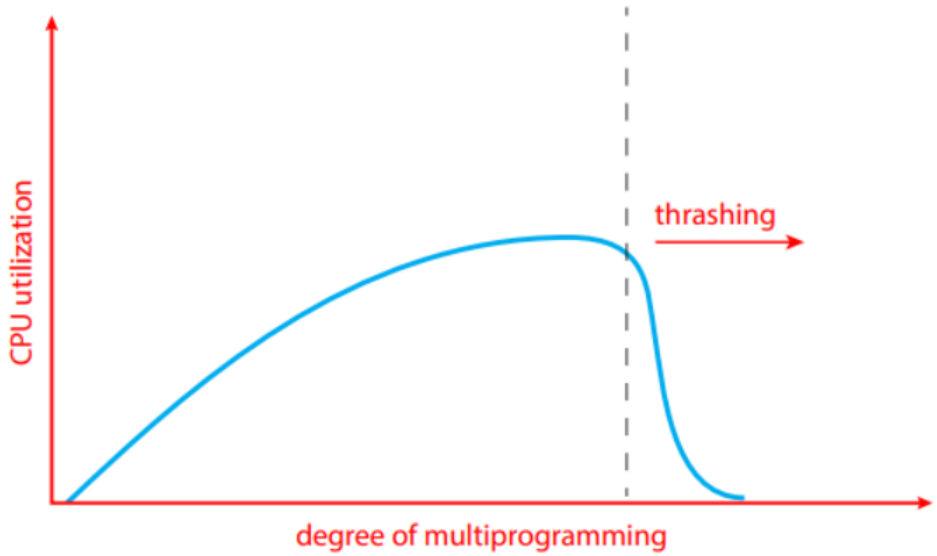

Figure 9: The Effect of Thrashing on CPU Utilization

The operating system monitors CPU utilization. If CPU utilization is too low, we increase the degree of multiprogramming by introducing a new process to the system. A global page-

replacement algorithm is used; it replaces pages without regard to the process to which they belong.

Now suppose that a process enters a new phase in its execution and needs more frames. It starts faulting and taking frames away from other processes. These processes need those pages, however, and so they also fault, taking frames from other processes. As they queue up for the paging device, the ready queue empties.

As processes wait for the paging device, CPU utilization decreases. The CPU scheduler sees the decreasing CPU utilization and increases the degree of multiprogramming as a result. The new process tries to get started by taking frames from running processes, causing more page faults and a longer queue for the paging device. Thrashing has occurred, and system throughput plunges. No work is getting done, because the processes are spending all their time paging.